OpenCDN: An ICN-Based Open Content Distribution System Using Distributed Actor Model

Arvind Narayanan, Eman Ramadan, Zhi-Li Zhang
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota, USA
Email: {arvind,eman,zhzhang}@cs.umn.edu

Abstract—**Building upon the results of recent works on understanding large-scale content distribution systems, we revisit CONIA, a Content-provider Oriented Namespace Independent Architecture for content delivery. The key idea of CONIA is to let any willing ISP or third party to participate as a content distribution network (CDN). In this paper, we propose a first step in the direction of an information-centric network-based open content distribution system (OpenCDN), that allows for better scalability, flexibility, and performance. In particular, we concentrate on the functions of the content store and routing elements (CSRs) that form the network substrate. We propose an actor-model driven programming model and a runtime system, which together we refer to as the OpenCDN platform. Using OpenCDN, content providers will have full control over building and managing the basic building blocks for the functionality of CSRs, and the flexibility on which content to cache, when to cache, and how to satisfy user requests.**

I. Introduction

While designing any system architecture, user experience is an important aspect to consider, due to its impact on user subscription and revenue. Nowadays, video content represents the majority of the downstream traffic, due to the available Internet speed, the wide use of social networks, along with the growth in mobile devices. Hence, more video content – by content providers and video streaming services – has been available to end-users. Netflix, YouTube, and Amazon Video, account for more than 55% of the downstream traffic of North America in 2016 [1].

No single server, or even a data center, has the processing capability, or the network bandwidth to serve all user requests for this scale. Hence, large-scale content providers (CPs in short) rely on *content distribution networks* (CDNs), which contain a set of proxy servers distributed geographically around the world. CDNs can either be: (1) homegrown owned by content providers, as in case of Google/YouTube [2], or (2) commercial CDNs, such as: Akamai [3], Level-3, and Limelight, used by Netflix [4], and Hulu [5]. CDNs are typically organized in a hierarchical structure ([2], [3], [4]), where edge servers lie at the closest tier to end-users, and the farthest (at the top of the hierarchy) represents the origin server, which has a copy of all content objects. Requests from end-users are directed to the closest edge server. If the requested content is not cached, some complex techniques, such as DNS redirections, HTTP redirection, and IP anycasting, are used to forward requests to higher layers in the hierarchy.

Coping with this large-scale content delivery, *information-centric networks* (ICNs), *e.g.*, NDN [6], [7], DONA [8], and idICN [9] (see [10], *e.g.*, for a survey of ICN architectures), have been proposed to address the limitations of today's host-oriented Internet architecture. All these designs share two basic design tenets: (i) *content is the first-class citizen that can be directly named and routed*; and (ii) *content storage should be part of the network substrate*. The second tenet allows for *in-network* caching and storage, that will be of utmost importance to handle the massive growth in content. Many of these designs (*e.g.*, NDN and idICN) still dictate a fixed name schema (either hierarchical or flat), and adopt a conventional "box-centric" view of network design – they focus on the design and functionality of individual network elements, instead of *network-wide* control and operations. Nor do they take advantage of emerging software-defined, programmable network elements, and network function virtualization.

Thus, we have proposed CONIA [11] to address the limitations of existing ICN architectural proposals, by advocating a *namespace* independent architecture, using open programmable content store and routing elements (CSRs in short) as part of the network substrate (as shown in Fig. 1). The key idea is to allow any ISP or third-party (*e.g.*, an existing CDN provider or even an end-user) to participate in content distribution by bringing their own generic and programmable CSR boxes (*i.e.*, cache servers with storage and compute power). CSRs are responsible for caching content, and providing the basic functionality for resource management and content delivery. In this paper, we propose an open control framework, which enables CP controllers manage and configure CSRs using *standardized* APIs, along with a runtime system, which altogether we refer to as the OpenCDN platform.

While designing OpenCDN for managing CSRs, the following key design goals are required: **(1) Modularity, compositionality and velocity**: this platform must support modularity among the basic building blocks, which allows them to continuously evolve to meet changing application requirements. However, updating certain blocks should not affect others. Additionally, some of the basic building blocks could be assembled together to form a new CSR operation, hence, the correctness of the system should be ensured under various operational contexts & environments. **(2) Scalability and**

availability: this platform also needs to support "auto-scaling" (via replication/parallelization) for individual blocks without affecting the system's performance. Moreover, the impact of a module's failure on others need to be minimized, with a fast recovery to meet the CP's quality of experience requirements even during failures. **(3) Performance**: the proposed platform also needs to take advantage of the available multi-core servers, multi-server clusters/clouds, and hardware accelerators (*e.g.*, DPDK [12], RDMA, NetFPGA [13], etc.). **(4) Mobility**: functions provided by this platform should have the flexibility to move from one server to another smoothly, which enables dynamic load balancing and fault tolerance.

One way to realize these goals is by using the "Actor model frameworks", such as Akka [14], Erlang [15], and Orleans [16], which are becoming increasingly popular in building large-scale distributed applications. However, existing actor model frameworks are not customized for multimedia content, and their performance is not optimized from the networking perspective as explained in the next section. This motivates us to customize an actor model-based runtime and programming model for CONIA's architecture. Our proposed OpenCDN platform uses the actor programming model, and provides an abstraction for content providers to manage and configure CSRs. It also includes inherited support for resource management, scalability, resiliency, and relocation/migration. The rest of this paper is organized as following: Section II provides the background and related work. In Section III, we decompose the operations of the CSR to identify a set of building blocks, and understand their behavior. In Section IV, we present our proposed OpenCDN platform, which leverages the actor model to realize the building blocks associated with CSR operations. The paper is concluded in Section V.

II. Background and Related Work

A. CONIA's Architecture

We have proposed *CONIA* [11] as a novel content (provider)-oriented, namespace-independent architecture for content delivery. No single namespace or schema can fit diverse content types, because content objects are complex and composite (*i.e.*, consist of other objects). For example, a movie consists of audio and video segments, which are often encoded in different bitrates. Thus, CONIA allows content providers to define their own namespace. Moreover, CONIA was designed to provide *a software-defined* paradigm for content distribution. Unlike existing ICN architectures, CONIA allows CPs to employ their own management policies.

Fig. 1 shows a schematic illustration of CONIA's architecture, which consists of three main components: 1) CP *controllers*, 2) *content store and routing* elements, and 3) *content players* at the end-user side. CP controllers are content-provider specific, responsible to provision and manage CSRs. Client content players can be generic or CP-specific software, which allow end-users to interact with CP controllers and CSRs. Upon receiving a user request, the CP controller sends a *Content MAP* (*i.e.*, metadata of the content – more specifically what to request & from where) to the content player situated

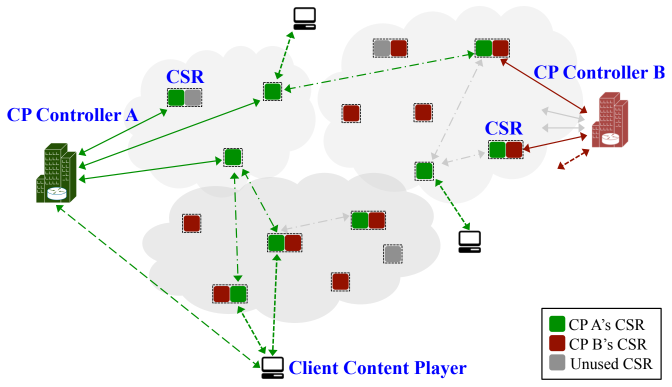

Fig. 1. CONIA's Architecture

at the end-user premises. CPs also define the control logic to dynamically specify how user requests should be handled, and install it in the corresponding CSR elements in the network substrate.

CSRs are generic, programmable resources for content delivery, which can be *shared* by multiple CPs. CSRs can be provided by ISPs, enterprise networks, CPs themselves, or third-party entities similar to commercial CDNs nowadays. CSRs are responsible for caching content, and providing the basic functions required for resource management and content delivery. CSRs also store CP-specific control logic to handle user requests and content objects, report the collected statistics to CP controller, and exchange data & health information with other related CSRs. CP controllers configure CSRs to install their control logic using an open control framework with *standardized* APIs. Upon receiving an object request, a CSR looks up and applies the corresponding control logic (see Section III for more details).

In Summary, CONIA is designed to handle the diversity and complexity of various content types. It provides each CP the flexibility to specify its own content management strategies such as cache management, load balancing, etc. to dynamically handle user requests, and optimize the content delivery process to meet users' quality of experience (QoE) expectations.

B. Actor Model Framework

Actor model frameworks such as Akka [14], Erlang [15], C++ Actor Framework [17] and Orleans [16] are becoming increasingly popular in building large-scale distributed applications. In such applications, actors are the basic building blocks or primitives. Each actor is an independent unit of computation, which contains its own private state, a mailbox (or a message queue) for accepting incoming requests, and a set of functions to realize a predefined behavior running on a single thread (see Fig. 2).

Actors are isolated from each other, and therefore do not share state information with one another. In other words, an actor can never change the state of another actor by itself. Actors interact asynchronously with each other using messages. Therefore, actors sending messages to other actors need not wait for a reply from other actors, thus non-blocking.

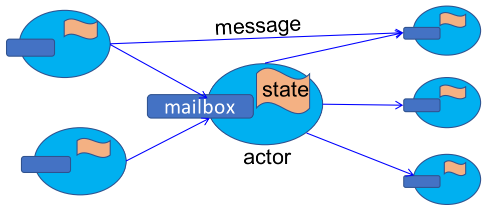

Fig. 2. The Actor Model

Incoming messages are received by the actor through its mailbox, which triggers the actor to run a set of functions that could either use or update the state information to finally produce an output. The output is then packed into a message, and sent to the next actor as defined by the behavior of the actor. Multiple actors can be running concurrently at the same time as if each is running in its own thread. Such model simplifies the programmability of the distributed system, and helps eliminate race conditions, which otherwise would require using locks and monitors.

C. *Related Work*

Existing actor model frameworks ([14], [15], [16]) are not customized for multimedia content. Especially their performance from the networking perspective is not optimized. For example DPDK [12] is not natively supported in Erlang, thus the actor messages use kernel networking stack. This motivates us to customize an actor model-based runtime and programming model for CONIA's architecture with high performance. In the following sections, we illustrate the basic CSR operations, and describe how the actor model framework can be used to implement them.

In order to accelerate HTTP and reduce network load, website administrators (or content providers) have for long relied on caching content objects on nodes that are able to fulfill end-user requests with reduced latency and better quality of experience. While there are plenty of such platforms, Varnish Cache [18] is one such system that can be configured by content providers to dictate how end-user requests are handled by specifying policies that decide what content to serve, where to fetch the content from and how the system should alter its internal state that affects future requests. Varnish partially achieves in what we try to accomplish with OpenCDN – which is to provide flexibility to content providers in driving the content delivery process. However, Varnish is built as one big monolithic system where are all operations are executed by one big logical caching server which makes it difficult to scale up. Moreover, Varnish does not natively support distributed caching instead relies on application developers to route requests to different Varnish instances by using load balancers (*e.g.*, HA Proxy).

III. Abstracting CSR Operations

As discussed in the previous section, the function of CSR includes caching content and handling user requests based on the control logic specified by the CP's controller. In doing so, we first decompose CSR operations into a set of basic building blocks or modules. Each of these building block has a well-defined behavior and specification of what input it accepts and the type of output it gives out. These blocks can either be stateful or stateless. In this section, we first walk through a scenario where – we identify the primitive building blocks associated with a CSR in fulfilling an end-user's request for a content object. Later, we dig deeper into the behavior of a few building blocks.

A. *Decomposing* FETCH *Operation*

Consider a scenario when a CSR receives a *FETCH* request for a content object from an end-user. In a nutshell, upon receiving such a request, the CSR parses the request, and checks if the requested piece of content is cached or not. If the requested object is cached, then the content is directly served by the CSR. Otherwise, depending on the control logic specified by the CP controller, the CSR can fetch the content remotely from an upper tier[1] CSR, or forwards the request to the CP controller (or origin).

Fig. 3 shows – in detail – the set of actions performed by a CSR to fulfill a content object (or a *FETCH*) request. First, the received request is sent to the `Parser` module to extract the *content_id*. This *content_id* is then passed to the `ContentLookup` module to check if the object is cached in the disk, in memory, or not cached at all.

Cache Hit. If it is in the memory, the object is sent to the end-user by the `ReturnObject` module. If it is in the disk, the object is loaded to memory through `InMemoryBuffer` module, then sent to the end-user.

Cache Miss. If the requested object is not cached, `ContentCacheMissLookup` module specifies the CP control logic defined to handle a cache miss event for each object or a group of objects. The control logic could either specify to forward the request to the controller, or to another CSR, or to fetch the object remotely from another CSR. In the latter case, the request is sent to the `RemoteCSRFetch` module, which picks a set of candidate remote CSRs. But instead of remotely fetching from multiple remote CSRs, the CP control logic may optionally use a load balancer to judiciously pick one remote CSR and fetch from it. To do so, the `RemoteCSRFetch` module sends the set of candidate remote CSRs to the `LoadBalancer` module and waits for a response. The `LoadBalancer` module queries the `SwarmMonitor` – a module that runs separately in the background to maintain health information of remote CSRs; and selects one CSR that is best according to the predefined CP-configured load balancing strategy. Once the `RemoteCSRFetch` module receives the response from the `LoadBalancer` module, it sends a message to the `PendingRequest` module. A record is then added to the internal state managed by `PendingRequest` module. It then waits for replies with content objects from

[1]Assuming CSRs are organized in a hierarchical structure, where the edge CSRs are the closest to end-users, and the upper tier CSRs are closer to an origin server. Origin servers have a permanent copy of the object collection.

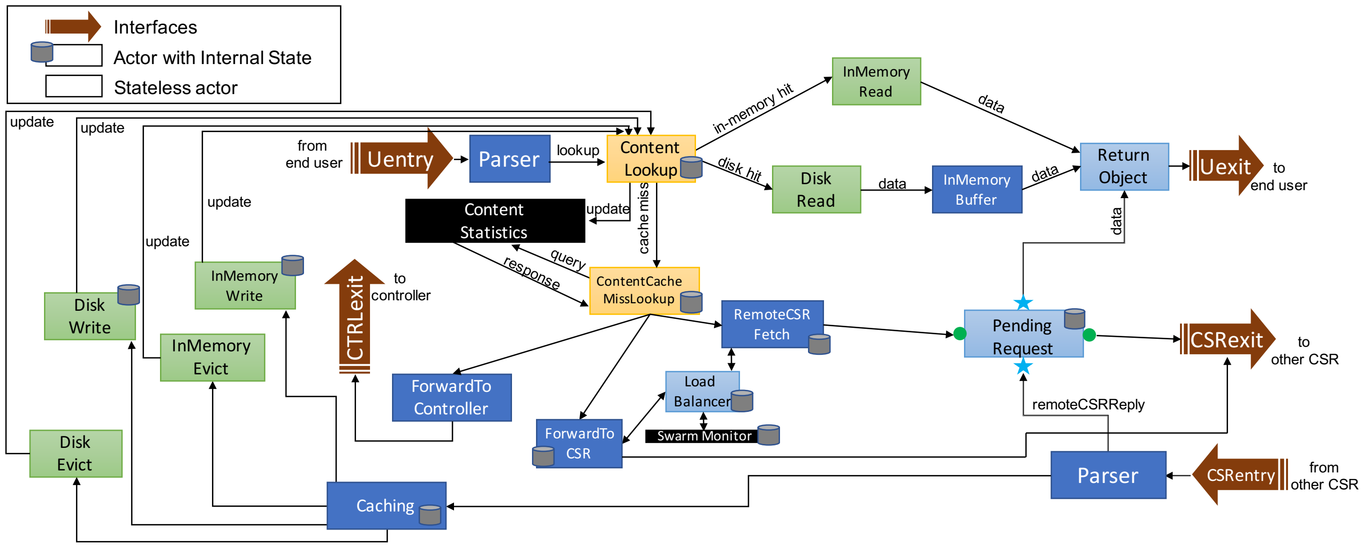

Fig. 3. CSR Operations

remote CSRs. All the modules are non-blocking and do not block other requests.

Pending Requests. Upon receiving a fetch request if the object is not cached, the state in the `PendingRequest` module is checked to see whether a request for the same object and remote CSR exists and is pending. If yes, the request will not be forwarded again, instead update the existing entry with the end-user information and the set of actions to be carried out when the requested content is received.

Remote CSR Reply. Upon receiving the object from a remote CSR (`CSRentry`), it is first parsed and two tasks are run in parallel: 1) serving end users, and 2) caching the object locally (if required). For the first task, the state for the object is loaded from the `PendingRequest` module which specifies the end-user(s) who requested this object and is forwarded to the `ReturnObject` module which replies to the respective end-user(s) with the content object. For the second task, the `Caching` module would check if the received content object should be cached locally by the CSR or not, if yes, whether to cache it in the memory or disk. If the disk or memory is full, the `Caching` module would also decide which object to evict. Depending on the actions to be taken, messages are sent to the appropriate modules in the path.

Memory/Disk Write. Given a content object and location, `InMemoryWrite` module writes it to the memory, and an update is sent to the `ContentStore` module, indicating what content object was added. Similarly, for the disk write operation as well, a similar flow can be seen. Due to space limitations, we skip the description of `InMemoryEvict` and `DiskEvict` modules.

B. Understanding the Building Blocks

After showing the flow of operations in fulfilling a *FETCH* operation, we now go deeper into understanding some of the building blocks. Each building block may need to store and maintain information in its internal state to realize its specific behavior. We illustrate with examples how such building blocks react to a specific input and how their internal state is managed and controlled.

ContentLookup. The purpose of this module is to check if a particular object is cached or not, and if it is cached provide more details on how to load it. The input of this module is in an object of type `CL_LOOKUP`. This input object contains two pieces of information: 1) content identifier or *content_id*, and 2) optional tags used to search the specific piece of content (*e.g.*, range of bytes, bitrate, language) or aides in the the lookup process (*e.g.*, DHT partition tags). To perform the lookup operation the `ContentLookup` module maintains a `ContentStoreTable` as its internal state. This table contains a list of content objects cached in the CSR. For each cached object, this table also contains information such as whether it is stored *in-memory* or *disk*, and the path to read the content object from, version information, timestamp when it was added to the CSR cache, etc. As a result, this module returns a `CL_LOOKUP` indicating whether the requested content object is `CACHED` (along with other information such as read-location) or `NOT_CACHED`. Based on whether it is a cache miss, a disk hit, or an in-memory hit, the resulting message is passed to the relevant module. The `ContentLookup` module also generates a `CSTAT_UPDATE` message and sends it to the `ContentStatistics` actor. The message contains the *content_id* and the status of lookup (*e.g.*, cache miss, in-memory hit, or disk hit). This module also accepts another type of input message, `CL_UPDATE` which is responsible for updating the contents of the `ContentStoreTable`. For example, as shown in Fig. 3, whenever content objects are written or evicted from the disk or memory, an update message (`CL_UPDATE`) is sent to the `ContentLookup` module – which consequently updates its internal state by either adding or removing entries to/from the `ContentStoreTable` to reflect the latest changes.

Parser. This module is responsible for parsing the incoming messages (raw byte stream) according to the parsing logic specified by the CP. It identifies the intent associated with each message, prepares the intent-specific object, and finally passes the object to the next relevant module. For instance in Fig. 3, we see there are two `Parser` based modules, one for the messages coming from the end-user interface (`Uentry`), and another for the messages coming from other CSRs (`CSRentry`). `Parser` is a stateless module as it does not need to maintain any state information to parse messages.

We can see that the realization of a CSR requires a set of building blocks or modules. Thus, a framework is required to provide an abstraction of these modules to enable CP controllers to configure them through a standard programmable API, regardless of how these modules are mapped to the physical resources of the CSR. Moreover, this framework also needs to support some required features such as fault tolerance, data replication, consistency, scalability, etc. without sacrificing the system's performance. Hence, we propose OpenCDN in the next section.

IV. Proposing OpenCDN

We propose an OpenCDN platform with the goal to foster an open and competitive content distribution ecosystem. The key idea is to enable any ISP or third-party (*e.g.*, an existing CDN provider or even an end-user) to participate collaboratively in content distribution by bringing their own generic and programmable CSR boxes (*i.e.*, cache servers with storage and compute power), thus becoming a CDN. In this section we outline the programming model and design choices for a runtime system that are critical to the OpenCDN platform.

A. Programming Model

We adopt the actor framework to implement the basic building blocks or modules identified for operating a CSR. This framework fits well in implementing our decomposed set of primitive modules where each actor encapsulates the behavior and internal state information of a primitive module. OpenCDN's programming model supports the following features:

State Information. Each module/actor maintains its own internal state. Depending upon the behavior of the actor, characteristics of the state information such as frequency of updates, consistency requirements across different servers or fault-tolerance levels may change. Therefore, we allow application developers to dictate what state information to persist, and when. They can explicitly define such persistence checkpoints in the actor's behavior, or configure it to periodically persist the internal state information. For example, in Fig. 3 the `ContentLookup` module maintains an internal state table named `ContentStoreTable`. This table contains the information about the cached content objects and their location.

Strong typed Language. A strongly typed language is considered to be a safer programming model mainly, due to the fact that a number of errors can be prevented at compile time itself, thus paving way for self-tested code. Actor interfaces in OpenCDN are all strongly typed. For example, `ContentLookup` module will only accept input objects of type `CL_LOOKUP` or `CL_UPDATE`.

Asynchronous Messaging Passing & Promises. Similar to the classical actor model, different actors interact with each other by asynchronous message passing. Therefore, an actor sending a query message does not wait for a response back instead gets a *promise*. A *promise* is a reference to a placeholder where a result of some task will eventually get stored. However, the querying actor would wait if it cannot proceed without getting the response from the reference pointed by the promise. The concept of a promise (or also referred to as a future) is popular among the parallel, concurrent programming, and distributed systems communities. For example, when a `ContentCacheMissLookup` module directs to fetch the content from a remote CSR, the subsequent module selects a candidate set of CSRs, and this set is forwarded to the load balancer. While the load balancer computes and checks the best CSR to remotely fetch the content, a reference to a promise is used and the process continues, instead of waiting for the result. The promise along with the content request is now forwarded to the `PendingRequest` module. Here, let's assume that an entry for the same piece of content is already present in its internal state, therefore, this module does not forward the content request to the remote CSR, instead it updates the existing entry in its internal state with the end-user's session details.

Actor Reference Addresses & Mailboxes. Actors are addressable, and use the address as a reference to send messages to each other. Incoming messages received by actors are queued in their mailbox. An actor will sequentially dequeue a message from its mailbox and process it one after the other. Multiple actors of the same type can run concurrently and separately as if they were separate actors. This enables autoscaling. For example, in Fig. 3 we find there are two Parser actors. One of them is connected to the messages received from users (`Uentry`) while another parser actor is getting incoming messages from other remote CSRs (`CSRentry`).

Location Transparency. Actors are location transparent. In other words, actors communicating with each other need not know the physical location except the address reference or the type of actor (if the reference is unknown). Similarly, even the applications using OpenCDN need not know the actor's physical locations.

Actor Monitoring. Similar to Erlang, an actor can spawn other actors (either locally or remotely across multiple CSRs) there by establishing a parent-child relationship. This is extremely useful to supervise actors. For example, a parent actor may know if a child actor has crashed and accordingly the parent actor can take appropriate action.

Code Changes at Runtime. An actor's behavior can also be changed at runtime (*i.e.*, hot code loading). This particularly is helpful in rapid prototyping.

B. Runtime System

OpenCDN is intended to not only run on a cluster of servers but across different geographic locations. Multiple instances of OpenCDN can also run on a single server concurrently. In order to support the programming model described earlier in a way that relieves application developers from making low-level configurations and managing the underlying services, OpenCDN's runtime consists of five subsystems:

Transport Services. Provides runtime services that enables message passing between actors. This includes setting up connections between different instances of OpenCDN runtime systems, re-use same connections to send or receive messages between different actors. The runtime also provides remote-actor-discovery and actor-message routing services by maintaining data structures that map actor reference addresses to OpenCDN runtime instances hosting actors. If an actor is moved from one server to another due to failure or other reasons, the actor-routing service will make necessary changes to its mapping tables. Runtime functions required to support DPDK and NetFPGA operations are part of this component.

Actor Lifecycle & Management Services. This component of the runtime manages the lifecycle of actors from the time they are created till they are closed. It is responsible to provide reference address at the time of actor initiation. Based on the relationship between different actors and their behaviors, this component provides actor placement services which decide the physical servers where actors will be hosted. Message passing between two local actors residing on the same server are carried out by this component. Services required to transparently migrate actors are also provided. Enabling actor mobility in a transparent fashion further helps in achieving dynamic load balancing and fault tolerance.

Resource Management. Runtime services required to monitor actors, physical servers and to wheel out load balancing strategies are provided by this component. Scheduling tasks corresponding to actors are managed by this component. It is important that fairness of resource usage is maintained and no actors take up resources indefinitely.

Application Logic Services. Execution of control logic code specified by application developers is managed by this runtime component. We specified earlier that in order to avoid race conditions, actors are single threaded. This component ensures that even after context switching, the resumed actor (or thread) will always at any point of time execute in a single thread only. Understanding state dependencies in the control logic of the applications is important so as to understand what actors or flows in the application logic can be parallelized.

Fault Tolerance, Data Replication & Consistency. Actors, OpenCDN runtime instances and servers may fail, messages may not get delivered and failure can happen in any form at any time. Handling them in a transparent manner without affecting other services and systems is crucial in building resilient systems. This relieves the programmer from managing failures explicitly. Similarly, critical data needs to be replicated for high availability and fault tolerance.

V. Conclusion

In this paper, we proposed a first step in the direction of an information-centric network-based open content distribution network architecture (OpenCDN). We provided an overview of CONIA's ICN architecture, and elaborated on the functions of the content store and routing elements (CSRs) that form the network and storage substrate of CONIA. We decomposed CSR operations to identify a set of primitive building blocks. We also studied the behavior of a few building blocks and how their state is used and managed. We highlighted the design of our proposed OpenCDN's an actor-model driven programming model and runtime system. This design allowed us to build a modular, scalable and resilient content distribution system. Moreover, it can also be leveraged to build other distributed systems with the same goals.

Acknowledgment

This research was supported in part by NSF grants CNS-1411636, CNS 1618339 and CNS 1617729 and a Huawei gift.

References

[1] Sandvine, "Global Internet Phenomena Report - June 2016."
[2] V. K. Adhikari, S. Jain, Y. Chen, and Z.-L. Zhang, "Vivisecting youtube: An active measurement study," in *INFOCOM, 2012*.
[3] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *SIGOPS*, 2010.
[4] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang, "Unreeling netflix: Understanding and improving multi-cdn movie delivery," in *INFOCOM, 2012*.
[5] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z. Zhang, "A tale of three CDNs: An active measurement study of Hulu and its CDNs," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 2012, pp. 7–12.
[6] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.
[7] "Named data networking," http://named-data.net/.
[8] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 181–192.
[9] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: Incrementally deployable icn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 147–158.
[10] B. Ahlgren and et al., "A survey of information-centric networking," *IEEE Communications Magazine*, July 2012.
[11] E. Ramadan, A. Narayanan, and Z.-L. Zhang, "Conia: Content (provider)-oriented, namespace-independent architecture for multimedia information delivery," in *ICMEW*, 2015.
[12] "Data Plane Development Kit," https://dpdk.org/.
[13] "NetFPGA," https://netfpga.org/.
[14] "Scala Akka," http://www.akka.io/.
[15] "Erlang," https://www.erlang.org/.
[16] "Orleans," https://research.microsoft.com/projects/orleans/.
[17] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting Actor Programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016.
[18] "Varnish Cache," https://varnish-cache.org/.